Synthesis and Optimization of Coordination Controllers for Distributed Embedded Systems

Pai H. Chou
Dept. of Electrical & Computer Engineering
University of California
Irvine, CA 92697-2625 USA
chou@ece.uci.edu

Gaetano Borriello
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350 USA
geatano@cs.washington.edu

ABSTRACT

A main advantage of control composition with modal processes [4] is the enhanced retargetability of the composed behavior over a wide variety of target architectures. Unlike previous component models that hardwire the coordination behavior either explicitly in the components or implicitly in the underlying model of computation, modal processes decouple component functionality and coordination protocols. Retargetability is achieved through the synthesis of *distributed* mode managers, which abstract away low-level synchronization and control communication details that would otherwise be exposed to the component designer. This paper presents an algorithm for the synthesis and optimization of distributed coordination controllers by computing an optimal projection of the global state space onto each processor. It not only minimizes interprocessor communication traffic for coordination but also reduces controller complexity by minimizing replication.

1. INTRODUCTION

In IP-based design, designers must be concerned with the *integration* of high-level components. A central issue in system integration is that components must have compatible protocols. Otherwise, a variety of "glue" mechanisms must be inserted for protocol translation.

Now, the term "protocol" applies to many levels. At the low level, designers are concerned with signaling on the pins and the datalink layer. Going up the protocol stack, another layer might be concerned with packetization or session establishment. What is usually overlooked is that above the *communication* protocol stack, the components must also agree to another kind of a protocol, namely *coordination*. Coordination governs what different components must do to perform a task collectively. For example, one component may perform one computation while the other component handles communication, or a set of components may run in low-power mode, which may necessitate other components to reconfigure accordingly.

The difference with coordination is that, unlike lower-level protocols, which are concerned with the *transport* of communication messages or signals, coordination protocols cannot be easily separated out with an API; instead, they are deeply ingrained in the component's functionality. As a result, they are the main cause for component modification and are becoming a serious obstacle to IP reuse.

The modal process model [5] was proposed to address some of these fundamental problems in IP reuse. It defines each component's coordination behavior *declaratively*, rather than imperatively; moreover, this representation for coordination is *composable*. The implication is that the coordination controllers required for system composition can be synthesized and optimized for each composition. Another key benefit is that this separation of policy and mechanism exposes many optimization opportunities for distributed target architecture. It is thus the goal of this paper to explore the synthesis and optimization of distributed embedded systems modeled with modal processes.

This paper first provides a brief review of modal processes and compares them with other approaches. Next, we propose a number of strategies commonly considered in partitioning. We then present an algorithm for optimizing interprocessor control communication in distributed architectures. The results from applying this algorithm are shown and discussed.

2. RELATED WORK

Today's component models can be classified many ways. They can be either platform-based or interface-based [13]. They can also be domain specific, and normally this means either data-dominated or control-dominated.

Platform-based components are designed for integration on specific implementation frameworks or infrastructures. These could be specific boards or busses; in the case of software components, the platform is usually the middleware or the operating system. Platforms are a fast way to assemble systems that can be operational shortly. In addition, platforms can be long lasting, and they may support system evolvability and incremental upgrade. They must standardize on protocols at multiple levels and often also fix architectural assumptions. While this may be desirable, the overhead may be prohibitive for small, cost-conscious or high-performance embedded systems.

Interface-based components are only known by their outside interfaces, which may be defined at several levels as well. They are not tied to specific platforms. Their interfaces are flexible or ab-

stract, such that one or more layers of the protocol stack can be replaced without affecting the component functionality [8]. However, synthesis is required to map the abstract constructs to those in the concrete platform before the system is operational.

Today's component models, whether platform-based or interface-based, commonly force designers to express coordination as an inseparable part of component functionality. This is especially true with most *control-dominated*, FSM-like models (StateChart and variants [15], Esterel [2], SDL [16]) or object-oriented models composed by method calls. Attempts to capture coordination using hierarchical state machines have resulted in overlapping states [10], which are difficult to understand and reuse. Another approach is to limit the components to specific classes of coordination protocols. Several abstract models exploit patterns in the behavior to enable optimizations. For example, synchronous dataflow [12] (SDF) coordinates by data dependency and has fixed input-output correlations; communicating sequential processes [11] (CSP) coordinates by rendezvous and has the property of speed independence. These properties enable the optimization of the coordination mechanism, usually in the form of a static scheduler that incurs no runtime overhead. However, these are domain specific solutions.

Several models have been proposed to address the problems with hardwired coordination protocols. Synchronizers [9] are a way of extracting synchronization policies from the objects and enabling their substitution. Mediators [14] bind method calls to events such that coordination changes need to be reflected only in the binding, rather than in the components. Both are software frameworks and thus they are not amenable to automatic, topology-specific optimizations or real-time scheduling. As a more abstract, interface model, DCCA [1] is perhaps the closest to our approach in that components are detached from their coordination behavior, which is stated in boolean algebra terms and synthesized as distributed controllers. They support a broadcast run-time environment, although broadcast may not be suitable for all architectures. Our proposed technique does not require broadcast and should be able to readily complement their work.

Our modal processes model takes an interface-based approach to high-level component modeling. It lets designers compose components by specifying correlations on the modes of operation. These correlations capture not just synchronization or invocation but also mirroring, exclusion, sequencing, and many combinations of patterns required for coordinating concurrent processes. Decoupling coordination and component functionality enhances modularity and enables synthesis and optimization of coordination controllers for any topology, and this would not be possible with platform-based approaches.

3. SYSTEM DESIGN FLOW

The designer creates a high-level model of the system by instantiating components and composing them. The components, modeled as modal processes, have *ports* and *modes* on their interfaces. Ports are for data composition while modes are for control composition. As in many models, ports are connected by channels; the unique feature about modal processes is that modes are related to other modes by constraints called abstract control types (ACT). These specify the rules on how one mode change can imply another set of mode changes, and they are the primitives for defining coordination protocols in a declarative way.

A mode change is called a *vote*, and it is generated by the process that owns the mode. ACTs constrain modes by effectively specifying a set of *transfer* relations on mode changes, as their purpose is to imply additional votes. Each ACT instance can be written as `actName`(*list of modes constrained*). Each ACT is sensitive to changes to a subset of its modes, and it triggers mode changes to another subset in response. One example of an ACT is `unify`, which is sensitive to all mode changes and propagates them to all other modes. Another example is `parent`(m, c[1..n]), which mimics a hierarchical state machine: when the superstate m exits (deactivates), all children c[1..n] must deactivate; activating any child c[*i*] activates m as well. Another example is `guardian`, a variation where the children are disallowed to activate when m is inactive. The reader is referred to [3] for a more detailed list of ACTs.

Once specified, the designer then maps this abstract model onto a target architecture. We assume the designer supplies the system topology for the purpose of system optimization; other details used for communication-level optimizations are outside the scope of this paper. The separation of behavior from architecture is one way we achieve better retargetability and expose optimization options.

Once both the abstract behavioral model and its mapping to the target architecture are obtained, then the synthesis tool implements the mechanisms needed to enable the system-level integration of these components. This paper deals with the realization of those mode relationships. When one component makes a mode change that affects the modes of components on remote processors, those components need to be notified of the mode change with a message, to which they may respond with acknowledgments or by synchronization. These control messages are dependent on not only the target architecture but also on how the designer maps the components to that architecture. By automating the synthesis of these control messages, we further eliminate low-level, architecture-specific design tasks that are exposed to the designer in other component models today.

4. STRATEGIES FOR DISTRIBUTED CONTROL

A mode manager must have access to a projection of the system configuration, where a configuration is a bit-vector representation of the component modes. Different projections are kept coherent by means of interprocessor communication. To take full advantage of distributed architectures, a partitioning algorithm should minimize communication among mode managers *and* minimize projection sizes. However, in general it is not possible to satisfy both goals, and the designer must make tradeoffs. This section considers a few strategies for replication vs. communication tradeoffs.

We will use a robot example to illustrate the concepts. We assume that the robot has five processes that are mapped onto three processors: joystick and pilot processes are on processor P1, bumper process on P2, and sonar and wheels processes on P3. Several partitionings of the mode manager are possible for a given process partitioning. The semantics of the actual ACTs should be of no concern.

4.1 Minimal replication

To minimize replication, one can host an ACT on the processor that hosts the most modes constrained by the ACT. This approach requires no replication of ACTs. For example, in Fig. 1 the ACT instance `uF`([F, P, Fd]) is placed on P3 because it hosts the majority of the modes constrained by `uF`, namely {P, Fd}. Ties are

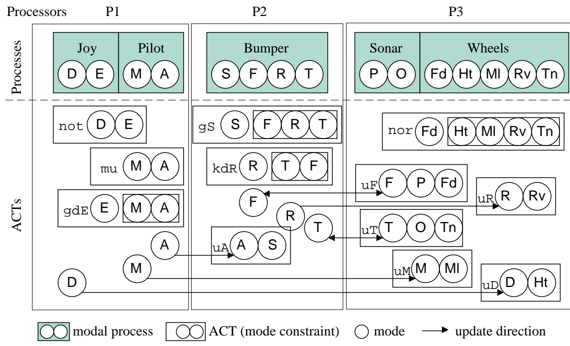


Figure 1: Minimum replication of ACTs.

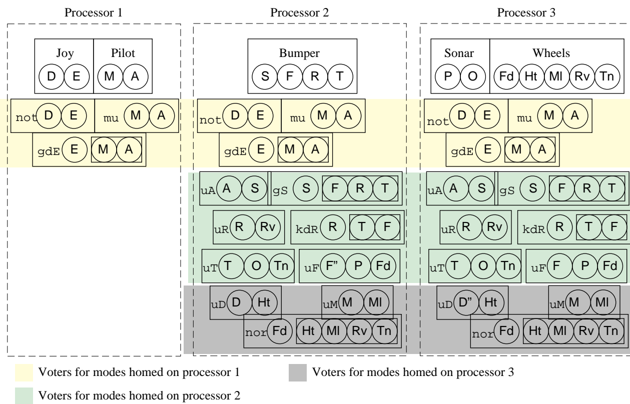


Figure 2: Partitioning of the robot example onto three processors with more ACT replication, requiring only original votes to be communicated.

broken arbitrarily, and replicated modes are those constrained by the ACT but not homed locally. No ACTs are replicated, and the choice of host for each interprocessor ACT minimizes the number of replicated modes. The receiver replies with an ACCEPT or DENY message.

Minimum replication incurs more communication than necessary, and thus is expensive for distributed topologies. Consider the case when switching to manual mode, P1 needs to communicate +M (*i.e.*, activate mode M) to P3, and −A (deactivate mode A) to P2. On P2, `uA` transitively propagates −A to mode S, and the ACT `gS` implies additional mode changes {−F, −R, −T}, which are in turn communicated to P3. It is slow because the communication and replies are propagated serially: P3 must accept these messages, reply to P1 and P2, and P2 in turn replies to P1. P2 also communicated {−F, −R, −T} unnecessarily, since it would have been possible to deduce them from the `gS` ACT locally from −S.

4.2 Maximal ACT replication

One attempt to reduce communication is to transmit only original mode changes initiated by the components, by maximizing local evaluation of transitive votes. Fig. 2 shows the same robot partitioned according to this scheme. P1 needs only its local ACTs: `not`(D, E), `mu`([M, A]), and `gdE`(E, [M, A]). No other ACTs or modes need to be replicated, because no other ACTs can vote on their modes directly or indirectly. Both P2 and P3 contain an entire copy of the ACTs, because both are able to vote directly or indirectly on the modes they host.

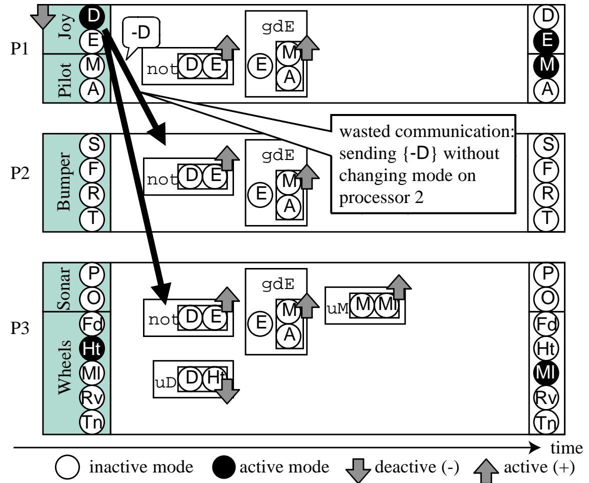


Figure 3: Example of wasted communication in the case of maximal ACT replication: −D is sent P2, even though in this case the vote is a don't-care and results in no mode change on P2.

A possibly surprising result is that maximal replication does not necessarily minimize communication, either. For example, in Fig. 3 suppose the system is initially in D mode when a mode change of +E is requested. P3 is told to go to M mode; however, the communication to P2 is wasted, because it changes only the replicated modes without affecting any modes hosted on P2. In other words, it fails to exploit temporal don't-cares and results in unnecessary communication.

4.3 Optimization strategy

Optimal partitioning is somewhere between the minimal and maximal replication schemes. Standard min-cut partitioning algorithms can be applied to determine the optimal cross-section bandwidth required in the worst case; however, the actual bandwidth may be smaller, as only a subset of votes is ever needed at one time. An optimal partitioning would project just enough ACTs and modes to eliminate wasted communication.

Fig. 4 shows an example of eliminating the unnecessary communication to P2 by moving the communication boundary to A. This not only obviates the need for replicating `not`(D, E), `mu`([M, A]), and `gdE`(E, [M, A]) (in the shaded region) on P2, but it also saves communication. On deactivating D mode, P1 transmits {−D} to P3 only, not to P2. P1 communicates with P2 only when the mode change affects mode A. This partitioning reduces communication *and* replication. We next present an algorithm that automatically finds this minimum cut.

5. MODE MANAGER PARTITIONING

This section presents an algorithm for minimizing communication as the primary objective and reducing ACT replication as the secondary objective. The main idea is to start with a full projection onto each processor, and then find a minimum cut in each projected ACT graph for reducing communication.

5.1 Representation

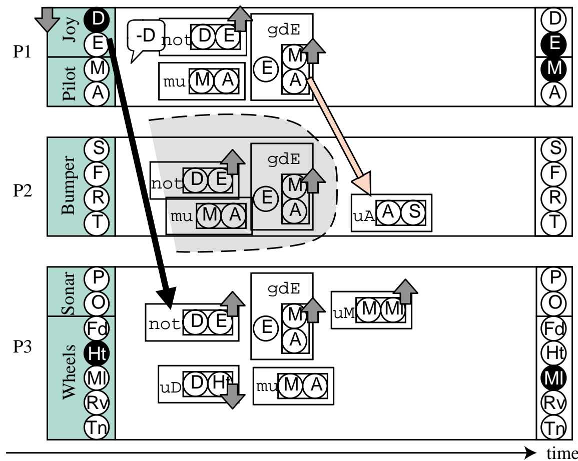
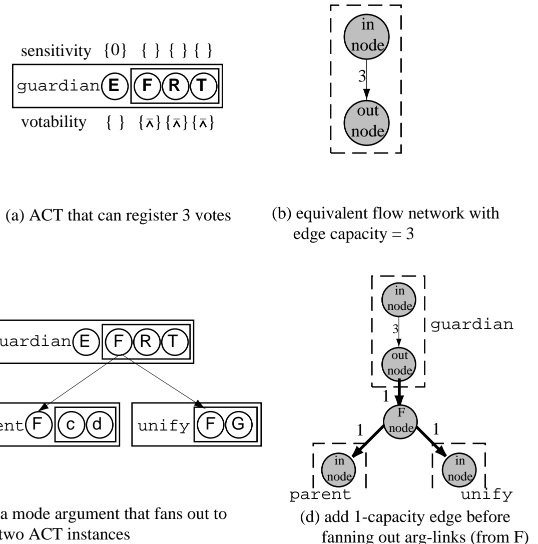


Figure 4: Example of ACT partitioning that minimizes communication bandwidth. Moving the communication boundary from (P1.D → P2.not) to (P1.D → P2.uA) eliminates unnecessary communication from P1 to P2 and reduces replication, since the three ACTs in the shaded region on P2 are no longer needed.



(a) ACT that can register 3 votes

(b) equivalent flow network with edge capacity = 3

(c) a mode argument that fans out to two ACT instances

(d) add 1-capacity edge before fanning out arg-links (from F)

Figure 5: ACT and the corresponding flow network.

The input to the mode-manager partitioning algorithm consists of the graph of ACTs and the *allocation* function α. An *ACT-instance graph* is a data structure that connects the ACT instances by the sensitivity of their arguments. The *sensitivity* of an argument is the set of vote values that causes the ACT to imply additional votes on other mode arguments. For example, in parent(m,c[1:n]), m's sensitivity is the singleton set $\{-\}$, or the deactivation value, because it causes the ACT to imply deactivation votes on all of c[1:n]. Similarly, sensitivity of c[1:n] is the singleton set $\{+\}$, or the activation value.

The graph can be defined as $G = (V,E)$, where V is the set of ACT instances and $E \subseteq V \times V$ is the set of directed edges connecting pairs of ACT instances. Each ACT instance constrains an ordered set of modes $r \in R$. We write r_i^v to denote the i-th argument of the ACT instance v. The ACT-instance graph is constructed by linking mode arguments by sensitivity. These edges are called *arg-links*. For example, in Fig. 5(c), guardian can vote $-$F, and both parent.F and unify.F are sensitive to deactivation values. As a result, we add an arg-link accordingly.

An allocation α is a function that maps a process $\pi \in \Pi$ to its processor number $[1,n]$, and we say that the process π is *homed* on processor $\alpha(\pi)$. Because a mode belongs to exactly one process, we overload the function α to map a mode to its processor ID, without ambiguity, namely $\alpha : M \rightarrow [1,n]$. The mode m is then said to be *homed* in process π on processor $\alpha(m) = \alpha(\pi)$. The algorithm projects the modes and ACT instances onto the individual processors according to the allocation. The output consists of the projected ACT-instance graphs for each processor and the interprocessor communication edges E_C.

5.2 Algorithm

The algorithm is shown in Fig. 6. It uses the MAX-FLOW MIN-CUT algorithm to determine the boundary for control communication that incurs the least cross-section communication bandwidth. The boundary also dictates which ACTs and modes need to be replicated. To solve this problem as an instance of max-flow min-cut, the algorithm constructs a flow network for each processor. A *flow network* is an abstracted representation of the ACT-instance graph, with local modes and local ACTs locked down as the sinks, and a set of remote modes and ACTs as the sources. Once the cut is determined, the vertices (modes and ACTs) on the source side remain remote, while those on the sink side must be replicated if not initially local.

5.2.1 Flow Network Construction

The corresponding flow network can be constructed based on the ACT instance graph. The number of votes an ACT can cast is represented as the edge capacity between a pair of nodes (in, out) in the network. For example, the ACT guardian(E, [F, R, T]) (Fig. 5(a)) can be represented as a pair of vertices with an edge of capacity 3, the number of modes that this ACT can register (Fig. 5(b)). All incoming arg-links (Sec. 5.1) in the ACT instance graph correspond to incoming edges to the in-node; all outgoing arg-links correspond to outgoing edges from the out-node. One difference is that if a mode argument fans out to several ACT instances (as in Fig. 5(c)), then the out-node must first be connected to a new node with a one-capacity edge before fanning out (Fig. 5(d)). Thus, a single voter cannot generate more than one vote. As a shortcut, arg-links between the same pair of ACT instances can be grouped into a single edge with the capacity equal to the number of arg-links.

5.2.2 Partitioning Loop

The same network is used for the partitioning of all processors, except different nodes are designated as sources and sinks. The sources and sinks are a mechanism for the algorithm to lock in vertices that are fixed in a partition. For notation, M^i is the set of modes homed on processor i. The set of vertices V consists of all the ACT instances, and V^i is the subset of those ACTs v that constrain only M^i. The set $X = V - (\bigcup_{i=1}^{n} V^i)$ contains ACTs that constrain modes across processors. For the purpose of max-flow min-cut on processor i, the nodes that correspond to the vertices in V^i are marked as the sinks, and they are all connected to a unique supersink by an edge with infinite capacity. The sources in the flow network are the remote modes $W(i) = \{m^j \in M^j, i \neq j\}$ whose processes can

ControlPartitioning(ACT instance graph $G = (V, E)$,
allocation $\alpha : \Pi \to [1, n]$) {
H := BuildFlowNetwork(G, α);
$X = V - (\bigcup_{i=1}^{n} V^i)$; /* *interprocessor ACTs* */
for $i := 1$ **to** n **do**
$N := H$; /* *make a copy* */
create new N.source s, new N.sink t;
foreach $w \in W(i) = \{m^j \in M^j, i \neq j\}$ **do**
N.edge := N.edge $\cup$ (s, w) with ∞ capacity
foreach $v \in V^i$ **do**
N.edge := N.edge $\cup$ (v, t) with ∞ capacity
$(S, T^i) :=$ MAXFLOWMINCUT(N, s, t);
/* T^i *captures the projection* */
end for
/* *find home for any unhomed interprocessor ACTs X* */
foreach $x \in X$ **do**
if $x \notin T^i \forall i$ /* *x is homeless* */
/* *pick processor j that hosts most of x's modes* */
$T^j := T^j \cup x$ such that $|x^j| \geq |x^{k \neq j}|$;
end foreach
return $T^1, \ldots, T^n$; /* *projections on processors 1...n* */
}

Figure 6: Partitioning algorithm for optimizing interprocessor coordination communication.

indirectly vote on M^i. The set $W(i)$ can be determined by simply following the arg-links in V^i in reverse in the ACT instance graph. A supersource node can be added to the flow network.

To compute the max-flow min-cut, several standard algorithms can be applied, including Ford-Fulkerson or the Edmonds-Karp implementation [7], which runs in $O(VE^2)$ time. The cut $c^i = (S, T^i)$ determines the projection: S is the set of vertices on remote processors, and T^i is the set for the local processor i. The cut set represents the votes that are registered and transmitted by remote processes or ACTs to the local processor i. Theoretically, the max-flow min-cut algorithm would allow votes to propagate in the backward direction, but in this construction, they are not possible because those communications would have been eliminated by ACT duplication. Note that T^i includes nodes for V^i by definition, but it can also include nodes that correspond to members of $V^{j \neq i}$ and of interprocessor ACTs X. While each intraprocessor ACT v^i has a well-defined home processor (namely i), the interprocessor ACTs $x \in X$ do not have a predetermined home processor, and at least one must be assigned. If a cut T^i does not include an $x \in X$, then x must be implemented on at least one of the other processors $j \neq i$. The partitioning algorithm is run n times for n processors.

Example

Consider the simple hierarchical FSM-like example shown in Fig. 7. The system has six modes, {B, C, D, E, F, G}. For the constraints, modes B and C are constrained by the `mutex` ACT `m1`([B, C]), and they are also constrained as `parent` ACTs `g1`(B, [D, E]) and `g2`(C, [F, G]), respectively. Suppose the designer partitions the modes into two sets, {B, D, E} and {C, F, G}.

To construct the flow network, each ACT instance is turned into a pair of vertices connected by an edge with the capacity equal to the number of modes. For example, each processor can vote on three modes ({B, D, E} on P1 and {C, F, G} on P2). They are shown

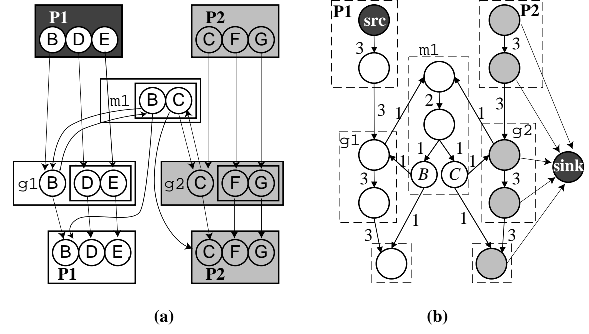


Figure 7: (a) An ACT-instance graph and (b) the flow network as input to MAX-FLOW MIN-CUT. To determine the incoming control communication bandwidth for P2, ACTs that are homed on P2 are marked as sinks in the flow network (light gray), and P1's source ACT is marked as the source in the flow network (inverted).

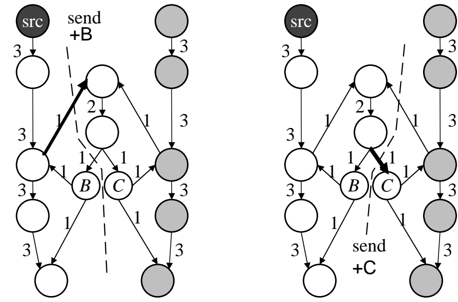


Figure 8: Two min-cuts for the example.

in the flow graph with a capacity-3 edge between the two vertices that are enclosed by the corresponding dashed box. The ACTs `g1` and `g2` are similarly constructed. The `mutex` ACT `m1` can register two votes, but the votes are individually fanned out. Therefore, additional fan-out nodes B and C are introduced to limit the edge capacity to one unit.

To determine the projection on P2, the sinks for the flow network are first chosen. They correspond to V^2's ACTs, namely `g2` and the two polar vertices for modes homed on P2. The source nodes for the flow network can be determined by following the edges in the ACT instance graph backwards from V^2 to all remote voters, and these are P1's source ACT. Two cuts have the same max flow of 1, as shown in Fig. 8. The cut shown on the left indicates that P1 should transmit activation of B to P2 for the evaluation of the `mutex` ACT `m1`, which is implemented on P2. The one shown on the right does not have a copy of `m1`; however, it assumes P1 transmits the activation of C to P2 after having evaluated `m1` on P1.

6. CONCLUSIONS

This paper presents the synthesis and optimization of distributed coordination controllers, or mode managers. Today's models require designers to sprinkle control messages for coordination, which can be architecture specific, error-prone, and difficult to change. By synthesizing and optimizing these coordination mechanisms, we enable design space exploration and enhance component reuse in control-dominated applications.

Unlike other automated partitioning tools, our technique is agnostic to *functional* partitioning, which may be user-guided or automated. Instead, we automate the optimization of *coordination*, which can be cleanly separated from component functionality, and this is made possible by the component model with modal processes.

This tool has been integrated into an embedded systems codesign framework [6], with specific support for control composition. Such a temporal approach is important for real-time constraints. Moreover, in low-power systems, components must be able to coordinate the power modes, in addition to coordinating functionality. Our work represents a first step towards enabling higher level design in this increasingly more complex problem space. Our future work includes not only addressing the issues specific to power coordination but also integrating our control-dominated coordination with formal dataflow models.

7. REFERENCES

[1] S. Aggarwal, S. Mitra, and S. Jagdale. Specification and automated implementation of coordination protocols in distributed controls for flexible manufacturing cells. In *Proceedings 1994 IEEE International Conference on Robotics and Automation*, volume 4, pages 2877–82, May 1994.

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[3] P. Chou. *Control Composition and Synthesis of Distributed Real-Time Embedded Systems*. PhD thesis, University of Washington, 1998.

[4] P. Chou and G. Borriello. Modal processes: Towards enhanced retargetability through control composition of distributed embedded systems. In *Proc. Design Automation Conference*, pages 88–93, June 1998.

[5] P. Chou, K. Hines, K. Partridge, and G. Borriello. Control generation for embedded systems based on composition of modal processes. In *Proc. International Conference on Computer-Aided Design*, pages 46–53, 1998.

[6] P. Chou, R. Ortega, K. Hines, K. Partridge, and G. Borriello. ipChinook: an integrated development environment for IP-based design of distributed embedded systems. In *Proc. Design Automation Conference*, pages 44–49, 1999.

[7] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1990.

[8] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[9] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Seventh European Conference on Object-Oriented Programming (ECOOP)*, pages 346–360. Springer-Verlag, July 1993.

[10] D. Harel and C.-A. Kahana. On Statecharts with overlapping. *ACM Transactions on Software Engineering and Methodology*, 1(4):399–421, October 1992.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] E. Lee and D. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[13] J. A. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *Proc. Design Automation Conference*, pages 178–183, June 1997.

[14] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[15] M. von der Beek. A comparison of statechart variants. In L. de Roever and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863, pages 128–148. Springer-Verlag, 1994.

[16] C. R. Z.100. Specification and description language. Technical report, ITU, General Secretariat, Geneva, 1989.